0018-8670/98/$5.00 (C) 1998 IBM

T Spaces

by P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford

With the creation of computer networks in the 1970s came the birth of distributed network applications. Since then, there have been many applications that spanned multiple machines, but in the last 20 years no one created a serviceable network middleware package for developing highly effective distributed applications, that is, until now. This paper describes the design and architecture of T Spaces, a project at the IBM Almaden Research Center that fills the network middleware void. T Spaces embodies the three main characteristics of a useful mechanism for network programs, namely, data management, computation, and communication. Since it has the potential to connect any program to any other program on a computing network, T Spaces is an ideal platform on which to build a global computing services platform where any program or system service is available to any other program or service. In addition, its small footprint and Java implementation make T Spaces an ideal platform for writing distributed applications for embedded and palm-top computers, thus forging a needed gateway from the emerging embedded and palm-top computers to established desktop and server computers.

Ever since the Intel 8080 microprocessor chip appeared in the Altair computer in 1975,1 microprocessors have grown far more powerful and far less expensive. Gordon E. Moore, co-founder of Intel Corporation, postulated that the capacity and capability of computers would approximately double every 18 months. This 1965 prediction still holds today for most types of computers. However, despite the tremendous strides made by the computing industry to make computers smaller, faster, cheaper, easier to use, and more plentiful, there is still a glaring void in the area of distributed computing. Programs running on one platform, e.g., the Apple Macintosh**, still do not have easy access to programs or services that are resident on another platform, e.g., the IBM Advanced Interactive Executive (AIX*) operating system. Although a physical network connection does exist between machines of these disparate platforms, a quality, high-function logical connection does not. The reasons for this lie mostly in the difficulties of writing software that can manage the vast differences in computing hardware and operating systems. Unfortunately, the diversity of computing platforms is growing as a result of the creation of a new segment of the computer market that consists of a variety of tiny computers.

There is an interesting side effect to Moore's Law. While the microprocessors leading the technology race become more powerful, the microprocessors that are two or more generations old become very inexpensive, which creates an opportunity for producing them in large volumes. Furthermore, we have reached a unique position in the evolution of computing where the computers from two generations ago are still powerful enough to be useful for general tasks. These microprocessors, embedded in a wide range of consumer appliances, are part of a market known as Tier-0 devices. This term derives from client/server/mainframe designs, where Tier-3 machines are mainframes, Tier-2 machines are file servers, and Tier-1 machines are desktop machines. The name Tier-0 device describes any computing device that is smaller than a desktop or laptop computer. Of course, this name describes PDAs (personal digital assistants), embedded computers, or practically any electronic device, since processor chips are becoming cheap enough to embed in practically everything.

The emerging market of smart (Tier-0) devices is bringing forth a new era of network computing. Growing numbers of smart devices that once operated in isolation are now being connected to networks where they are starting to exchange information with one another and with larger network computers. However, there is a problem. Most of these devices, from automobile computers to atmospheric controllers for buildings to personal digital assistants, like their desktop ancestors, were not designed to communicate with platforms other than their own. They do not share a common data format, a common data schema, or even a common computing platform. Bob Metcalf, creator of Ethernet, is quoted as saying that the power of a computer increases with the square of the number of computers to which it is connected. It is clear to see that for information-sharing purposes, more connectivity is better, but the main challenge is building the connections. On the surface, connecting computers seems simple, but what looks like a simple problem is actually an immense one, all based around one theme: incompatibility. Moreover, with a new generation of even smaller computers entering the scene, the incompatibility problem is worsening.

The challenge, as we see it, is to find a way to enable high-quality communication between all computers, large and small. The benefits of succeeding would be enormous. By eliminating the numerous isolated islands of information and creating a single continent of knowledge, we can significantly reduce redundant tasks and overlapping systems. One example of an archipelago of information islands is a hospital. In many hospitals, even though patients sign in at the admissions desk, they also typically sign in at every department they visit, restating their vital information. Then, each health-oriented machine runs in isolation, generating either paper output or computer output generally incompatible with the rest of the hospital computers or hardware.

What is needed to solve this problem is network middleware--a software communication package that facilitates communication between programs. However, how does one create such a general-purpose solution? In the past, the enabling software--the system software--evolved one layer at a time. In the beginning there was no system software at all; in the very first computers, application programs contained all the logic to drive the bare (computing) machines. Over time, operating systems evolved, and then more system software layers evolved as common components were factored out of the applications and were made standard system components. What common elements, then, can we take from existing systems to create a new common software layer that will enable communication between all computers?

Network middleware requirements. To demonstrate what characteristics we might look for in our network middleware package, we take the example of the local area network (LAN) in an automobile. Today, an automobile has an average of 20, mostly independent, computers, each having a dedicated task. Although these computers currently have communication via ad hoc channels, they lack a unifying data management solution, such as that provided to Tier-2 and Tier-3 applications by relational database systems. Over time, once a unified communication network is available, the automobile computers will be able to easily incorporate new data from other computers to do their jobs better. In addition, the automobile manufacturer and mechanic can operate more efficiently using information generated by the automobile (e.g., brake pad wear readings, counts of engine redline crossings). Consider the task of building a unified communication platform for these 20 computers to communicate their operating status to their peers and to the central external communication module. How would one do this? There is a whole set of questions to answer:

- o How do the automobile computers obtain the network address of every other computer in the automobile?
- o What common data format do they use?
- o How do they resolve format differences, such as big or little Endian integers and different string formats?
- o How do they establish data transfer sessions?
- o What do they do if the receiving computer is busy?
- o What amount of information does each computer need to save, and for how long?
- o How does each computer figure out what all the other computers want to hear?

What characteristics should the network middleware have to meet the needs of this application? First, the communication software must have a footprint small enough to fit in each of the embedded computing devices. Second, it must be flexible enough to adapt to new data types. When a new message is added to the device community, it must be accepted by the rest of the group (or at least tolerated). In addition, it must tolerate new devices being added to the system. Third, it should provide a messaging service--"e-mail" for the various devices and processes running on the automobile LAN. In a real-time environment, such as an automobile, it is crucial that messages can be sent asynchronously, because the intended recipient may be too busy to answer or acknowledge them immediately. Fourth, the network middleware must support anonymous communication. By design, an automotive device does not send a message to another specific device, but instead sends to the group, expecting those interested devices to listen. Thus, communication is not point-to-point, but is closer to "multireceive" (as opposed to multicast), where devices register which types of messages they are interested in. Finally, the network middleware should have a data repository for storage of simple data. The main controller unit must track all of the events in the automobile for some period of time, which implies that there must be some storage and some search or query capability to find records in the stored data.

The automobile scenario is just one example of a set of devices that must hare information without predefined conventions, but there are many other similar examples in the areas of home LANs, office equipment LANs, hospital LANs, and aircraft LANs.

Previous attempts at network middleware. Previous attempts at solving the network middleware problem have not focused on the whole problem but instead have addressed some of the parts, namely data representation and client communication. The data representation problem has been addressed by CORBA** (Common Object Request Broker Architecture), a standard proposed by the OMG (Object Management Group) standards organization and relational database systems.2 The client communication problem has been addressed by Tuplespace systems.

The data representation problem. Building a general point-to-point solution that would allow programs in any language on any platform to communicate with any program on any other platform is genuinely difficult. The difficulty of this task increases for each new type of programming language, operating system, and hardware platform. Instead, a more realistic solution would be to either create a common data language that all programming language objects could translate to, or create a common data store that contained all data that needed to be shared. The common data language solution is the approach taken by CORBA. Programs map their data structures into the CORBA IDL (interface definition language) and then allow other programs to reference those data structures remotely through a proxy, using remote method calls.

Whereas the focus of CORBA was on creating a single interface to programming language (object) data, relational database systems, such as IBM's DB2* (DATABASE 2*),3,4 Oracle**,5,6 and Sybase**,7,8 used a different approach, which was to store the data in a database-specific format, decomposing the program data into the primitive atomic types used by the database system. The database system offered associative (rather than direct) addressing, which had the effect of disconnecting the data from the application. Although this effect had some beneficial results--programs and data could evolve separately--the database system often could not adequately represent the program data given its restrictive sets of datatypes and its inability to express complex relationships between entities. Although object database systems, such as ObjectStore,9,10 Versant,11,12 and O2,13,14 offered a more programming-language-centric form of storage for data than relational systems, they were too platform-specific in the data representation. Thus, they offered little help in solving the cross-platform communication problem. Finally, although there were several research projects, such as SMRC,15,16 QuickStore,17 and Postgres,18 and some products, such as IBM's UDB (Universal Database), Oracle V8**, and INFORMIX**, that combined notions of relational and object databases, they also did little to address the cross-platform communication problem.

In all fairness, the purpose of the database products and research projects was to expand the query power or the data expressibility of the database systems, not their ability to provide mechanisms for complex program interactions across multiple platforms.

The client communication problem. The problem of high-level distributed communication in a dynamic and diverse network has received little attention. Although not originally designed for this purpose, Tuplespace systems have been shown to provide many useful facilities for client communication.

Tuplespace systems evolved differently than database systems. Operating more like global communication buffers than data repositories, Tuplespace systems have always played the role of traffic cop for data flowing from one process to another in parallel and distributed systems. They have mostly functioned as global communication buffers that impose no schema restrictions. Thus, Tuplespace systems are tailor-made for distributed programming where a general data delivery mechanism is needed. Most research in Tuplespaces has been in the parallel programming community; however, in the early 1990s several systems used Tuplespace as the basis for distributed communication mechanisms.

Laura19 is a language that provides a coordination mechanism for "open distributed computing," which is defined to be those systems that are "dynamically composed from nondedicated hardware and software components." They use Tuplespace as a brokering mechanism for service providers and clients. Their system provides the basic tools to create distributed systems; however, it does not provide fault tolerance, a data repository, or tools for large-scale systems such as access control.

The ObjectSpace20 project added C++ object orientation to the standard Linda model. In ObjectSpace, any C++ object can be a tuple. They augmented the standard tuple matching algorithm to allow a template and a tuple to match if the type of the tuple is an instance of the type of the template--in which case, the object in the tuple can implement the object in the template. Adding object orientation was a good step toward making

Linda more flexible for distributed applications; however, this is just a starting point.

Recently, the combination of Java** and Tuplespace has received renewed interest; projects such as Jada21 and JavaSpaces**22 combine the two. Jada is a Linda implementation that is used to provide basic coordination for PageSpace,23 a high-level coordination system. JavaSpaces, currently under development at Sun Microsystems, is designed to provide "distributed persistence" and aid in the implementation of distributed algorithms. The system allows arbitrary Java classes to be communicated as tuples and made persistent through Tuplespace. Transactions are provided for Tuplespace integrity, and a facility for notifying a process when a tuple is written to a Tuplespace is provided instead of the standard blocking read and take operations. JavaSpaces provides a simple transactional data repository and communication mechanism. For simple applications, the functionality provided by Jada and JavaSpaces is similar to that of T Spaces, since they all derive from the same basic Tuplespace ancestor. However, T Spaces adds more advanced operator and database functionality so that it may support more complex applications. In addition to these projects, we have been informed that computer science departments are now assigning programming projects that feature Tuplespace implementations in Java.24

Although the similarity between Tuplespace and databases has long been recognized, surprisingly few projects have explored the connections between the two subjects. Persistent Linda (PLinda)25 was the first project to add database functionality, including transactions and a simple query and join engine, to Linda. The primary goal of the project was to provide a unified environment for competitive concurrency control (shared resource access) and cooperative parallelism (processes working on different pieces of a large problem). The system was used to provide distributed shared memory and transactions for a programming language, Griffin;26 however, the system was mainly targeted for parallel applications.

Tuplespace is a good starting point for a distributed communication system because it provides communication, synchronization, and a simple data repository in one framework. Yet, these systems did not attract much interest. Although they provided some heterogeneity, portability, and interoperability, they never were able to provide a solution that addressed all the issues well.

The solution. In this paper, we describe IBM T Spaces, a new network middleware system. It uses the Tuplespace model27-29 of interaction for building a globally visible communication buffer. It then extends the power of Tuplespace with database features traditionally found in large (Tier-2 and Tier-3) enterprise database systems. Furthermore, being implemented in Java, it inherits the ability both to run on virtually any platform and to download new datatypes and new functionality dynamically. The combination of the Tuplespace communication model and sophisticated data management features with the flexibility, portability, and strong typing of Java results in a framework that provides at once a lightweight database, an extensible computation environment, and a secure, yet easy-to-use communication layer. As we will show, this framework can manage exotic, free-format data across a wide range of devices, platforms, and tiers, all in a network-accessible manner. This paper is organized as follows:

The next section introduces Linda, the original Tuplespace system on which T Spaces is based. The third section presents an overview of T Spaces. Then a detailed description of the architecture of both the client and server portions of T Spaces is presented. The fifth section provides a developer's view of how T Spaces might be used in some applications. The last section concludes the paper and presents plans for future work.

Tuplespace

A Tuplespace is a globally shared, associatively addressed memory space that is organized as a bag of tuples (defined below). The Tuplespace concept was originally proposed by Gelernter in References 27 and 29 as part of the Linda coordination language. The combination of a standard sequential computation language (such as C or FORTRAN) and a small number of Tuplespace communication primitives produces a complete parallel programming language (e.g., C-Linda or FORTRAN-Linda).

The basic element of a Tuplespace system is a tuple, which is simply a vector of typed values, or fields.30 Templates are used to associatively address tuples via matching. A template (or anti-tuple) is similar to a tuple, but some (zero or more) fields in the vector may be replaced by typed placeholders (with no value) called formal fields. A formal field in a template is said to match a tuple field if they have the same type. If the template field is not formal, both fields must also have the same value. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field. Table 1 shows some simple tuples and templates.

A tuple is created by a process and placed in the Tuplespace via the write primitive. Tuples are read or removed with read and take primitives, which take a template and return the first matching tuple. (Note that, because the space is unstructured, the choice among multiple matching tuples is arbitrary and implementation-dependent.) Most Tuplespace implementations provide both blocking and nonblocking versions of the tuple retrieval primitives. A blocking read, for example, waits until a matching tuple is found in the Tuplespace, whereas a nonblocking version will return a "tuple not found" value if no matching tuple is immediately available.

o Distinguishing features. Tuplespace provides a simple, yet powerful mechanism for interprocess communication and synchronization, which is the crux of parallel and distributed programming. A process with data to share "generates" a tuple and places it into the Tuplespace. A process requiring data simply requests a tuple from the space. Although not quite as efficient as message-passing systems, Tuplespace programs are typically easier to write and maintain, for the following reasons:
o Destination uncoupling: Most message-passing systems are partially anonymous: it is not necessary for the receiver of a message to identify the sender, but the sender must always specify the receiver. The creator of a tuple, however, requires no knowledge about the future use of that tuple, or its destination, so Tuplespace communication is fully anonymous.
o Space uncoupling: By using an associative addressing scheme for tuples rather than a physical one, Tuplespace is able to provide a globally shared data space to all processes, regardless of machine or platform boundaries.
o Time uncoupling: Tuples have their own life span, independent of the processes that generated them, or any processes that may read them. This independence enables time-disjoint processes to communicate seamlessly.

Tuplespace extends message-passing systems with a simple data repository that features associative addressing. Conceptually it ranks above a pure message-passing system in terms of function but far below relational database systems, since most implementations do not include transactions, persistence, or any significant form of query facility.

The fundamental advantage of a Tuplespace system is flexibility. Lacking a schema, a Tuplespace does not restrict the format of the tuples it stores or the types of data that they contain. Since the needs of modern distributed systems primarily revolve around flexibility, Tuplespace is an obvious choice. The scalability of a Tuplespace system is provided by the complete anonymity of tuple operations. Neither server nor client has to keep track of connected processes. Time uncoupling is provided by the database-like character of the Tuplespace, whose lifetime is independent of any client process. Furthermore, the simplicity of a Tuplespace system enables it to run in a limited environment. Finally, the self-defining nature of tuple communication allows a significant degree of interoperability and extensibility.

Clearly, Tuplespace provides an interesting starting point in our search for network middleware for Tier-0 computing. However, a traditional Tuplespace implementation is not enough. The matching algorithm will not work in a heterogeneous environment, where different platforms may be using different type systems. Furthermore, although Tuplespace loosely resembles a simple database system, the Tuplespace data are not necessarily made persistent, and the associative addressing is simply too primitive for many interesting data management problems. Finally, without any structure imposed on the tuples in the space, a traditional implementation will not easily scale to support efficient queries on the large numbers of tuples that may be generated by a network of Tier-0 devices. In the next section we describe how our combination of Tuplespaces, Java, and advanced database technology produces a lightweight, flexible, network middleware solution.

T Spaces overview

T Spaces is network middleware for the new age of ubiquitous computing. It is implemented in the Java programming language, and thus it automatically possesses network ubiquity through platform independence, as well as a standard type representation for all datatypes. It extends the basic Linda Tuplespace framework with real data management and the ability to download both new datatypes and new semantic functionality.

The salient features of the T Spaces system are:

- Tuplespace operator superset: T Spaces implements the standard set of Tuplespace operators: read, in (take), and out (write). In addition, it includes both blocking and nonblocking versions of take and read, set-oriented operators such as scan and consumingscan, and a novel rendezvous operator, rhonda, explained later.
- Dynamically modifiable behavior: In addition to the expanded set of built-in operators, T Spaces allows new operators to be defined dynamically. Applications can define new datatypes and new operators that are downloaded into the T Spaces server and used immediately. This is in contrast to relational database systems that have limited datatype support and limited dynamic function (usually in the form of triggers).
- Persistent data repository: T Spaces employs a real data management layer, with functions similar to heavyweight relational database systems, to manage its data.31 T Spaces operations are performed in a transactional context that ensures the integrity of the data.
- Database indexing and query capability: The T Spaces data manager indexes all tagged data for highly efficient retrieval. The expanded query capability provides applications with the tools to probe the data with detailed queries, while still maintaining a simple, easy-to-use interface.
- Access controls: Users can establish security policies by setting user and group permissions on a Tuplespace basis.

o Event notification: Applications can register to be notified of events as they happen in the T Spaces server.

T Spaces is appropriate for any application that has distribution or data storage requirements. It can perform many of the duties of a relational database system without imposing an overly restrictive (and primitive) type system, a rigid schema, a bulky user API (application programming interface), or a severe run-time memory requirement. In a sense, it is a database system for the common everyday Tier-0 computer--one that does not generate complex SQL (Structured Query Language) queries, but one that needs reliable storage that is network accessible.

T Spaces: The design, architecture, and implementation

Following the Tuplespace model, a T Spaces client communicates with a T Spaces server via tuples--ordered vectors of fields, each describing a type and a value. A T Spaces client issues operation tuples that perform a variety of functions. Figure 1 shows a high-level view of two clients communicating via a T Spaces server.

First, client 1 issues a write call to insert the <test1> tuple into the Tuplespace. The <test1> tuple is sent to the T Spaces server, where it is stored in the T Spaces tuple database. Then, client 2 issues a read query, specifying <test1> as the query tuple. The query tuple is sent to the server and used to query the server tuple database. The <test1> tuple is found, and since it is a read query and not a take, a copy of the tuple is returned to client 2.

The following subsections explain the details of the T Spaces client/server design, architecture, and implementation. They elaborate on the details of what goes on behind the scenes of the seemingly simple interaction shown in the above example.

T Spaces clients. The client API of T Spaces consists of a simple interface of tuple operations, mostly oriented toward reading and writing tuples. T Spaces users find that this simple interface is sufficient for a wide variety of applications; however, for the sophisticated user, T Spaces provides some advanced functionality as well, such as advanced query operations, security and authorization operations, event registration, and dynamically definable operators. Here we elaborate on the basic and advanced operations that are available to users, along with (some of) their underlying implementation. In the later subsection on the T Spaces server, we cover the topics of security, events, and dynamically defined operators, along with other server topics.

Basic T Spaces tuple operations. The basic tuple operations are write, take, and read. Write stores its tuple argument in a Tuplespace. Take and read each use a tuple template argument that is "matched" against the tuples in a Tuplespace (see the next subsection for the matching algorithm). Take removes and returns the first matching tuple in the Tuplespace, whereas read returns a copy of the matched tuple, leaving the Tuplespace unchanged. If no match is found, take and read each return the Java type null and leave the space unchanged. Blocking versions of these, waittotake and waittoread, are provided, which (if no match is found) block until a matching tuple is written by another process. (Linda programmers will recognize the semantics of these primitives as out, inp, rdp, in, and rd.) T Spaces also extends the standard Tuplespace API with the operations scan, consumingscan, and count. Scan and consumingscan are multiset versions of read and take, respectively, and return a "tuple of tuples" that matches the template argument. Count simply returns an integer count of the matching tuples.

In addition to these basic operators, we have implemented a new operator, rhonda. The rhonda operator takes a tuple and a template as arguments and atomically swaps them with a matching template and tuple from a rhonda executed by another process. If process 1 executes rhonda(<"A">,<String>), for example, which writes the tuple <"A"> and requests any tuple with a string value, and process 2 executes rhonda(<"B">,<String>), then process 1 will receive the tuple <"B">, while process 2 will receive <"A">. This is useful for atomic synchronization, such as atomically returning a "ticket" to check the status of a service request when a client issues the request.

Tuple matching. An extended Linda tuple-matching algorithm is used to determine whether a tuple in a Tuplespace satisfies a tuple retrieval request (read, take, etc.). In the standard (Linda) case, the template is simply a tuple, with one or more formal fields. (Recall that a formal field has a type, but no associated value.) A tuple matches the template when all of the following conditions hold:

1. The tuple and template have the same number of fields.

2. Each of the fields of the tuple is an instance of the type of the corresponding field of the template.

3. For each nonformal field of the template, the value of the field matches the value of the corresponding tuple field.

Condition 2 simply extends the Linda notion of exact type equivalence to an object-oriented notion of type equivalence.

Advanced tuple matching. The basic structural matching described above is similar to that used in most Tuplespace systems. However, in order to provide greater flexibility and database-like functionality, T Spaces uses a more object-oriented approach that offers more advanced matching options. These options are object compatibility, named field matching, and query semantics.

Object compatibility. For a tuple to match an object compatibility template (known as a Subclassable Tuple template), the type of the tuple in the space must be a subclass of the type of the template. In this way, a programmer can treat tuples as objects, customizing them to specific tasks. For simple applications where only structural equivalence matching is needed, the Tuple class of the T Spaces is final (i.e., it cannot be subclassed), and thus, when it is used as a tuple or a template, it will only match other tuples or templates of type Tuple. Table 2 shows some sample tuples and templates.

Named fields. One of the differentiating features of T Spaces that distinguishes it from conventional Tuplespace systems is that it builds an index on each named field in a tuple. This feature enables clients to request tuples based solely on values in fields with a given name, regardless of the structure of the rest of the tuple (or even the position of the named field within the tuple). For example, an index query of the form ("foo",8) will select all tuples (of any format) containing a field named "foo" with the integer value 8. It is also possible to specify a range of values to be found in the index.

Queries: Tying it all together. The current implementation of T Spaces provides four types of queries: Match, Index, And, and Or queries. A Match query performs structural or object compatibility matching (depending on its argument), whereas an Index query performs a named-field query. And and Or queries can be used to combine these other queries and build complex query trees. Examples of the inserted data are shown in Figure 2, and queries for those data are shown in Table 3.

The explanation for the numbered queries in Table 3 is as follows:

1. Query 1 is a regular structure Match query, where the query values are fed directly into the read operator. In this example, the query will return the first tuple of the form <"Superman", 75, Rock("Kryptonite")>.

2. The Match query's functionality is similar to the regular structure Match query, but it takes a query tuple as input. In this example, the query will return all tuples of the form <"Superman", 75, Rock>, where the values for the third parameter, Rock, can be any valid Rock value.

3. The Index query is either an exact match or a range. In this example, it is an exact match on the value "Spiderman." This query will return all tuples of any structure that have a Superhero field of the String type, with the value "Spiderman."

4. The fourth one is an example of an Index query using a Range predicate. This query will return all tuples of any structure that have a Superhero name in the range of "A" through "L."

5. The fifth one is an example of an And query. And and Or queries can be arbitrarily nested and used in any combination with other query types. This query will return all tuples of any structure that have a name in the range of "A" through "L" and an age in the range of 10 through 30.

6. Query 6 is an example of an Or query and is left as an exercise to the reader.

Currently queries are restricted to single spaces. However, since tuples of any shape or size are allowed to co-exist in a single space, this is not overly limiting. In addition, having multiple tuple types in the same Tuplespace provides interesting parallels with relational database systems. For example, an Index query over a single column is similar to a select predicate applied to the result of a join of N tables (where N is the number of distinct tuple types taking part in the query).

Note that a common operation in relational database systems is a join. In T Spaces terminology, the join is a user-defined, not a built-in, operator (scan the tuples with the specified column names, perform a "query" operation, and a merge) that directly matches existing tuples. Other relational database operators, such as aggregation, group by, union, etc., could easily be added as well.

Client implementation. For ease of exposition, we describe the client implementation in terms of the client being connected to a single T Spaces server. The implementation, however, allows the client to concurrently issue requests to multiple T Spaces servers.

All communication between the client and T Spaces server is completely nonblocking. If a client thread issues a blocking request, it is blocked in the client-side library after sending the request and is awakened when the response arrives. In this way, multiple threads in the same Java Virtual Machine can share a Transmission Control Protocol/Internet Protocol (TCP/IP) connection to the T Spaces server. The design allows multiple connections; however, the current system creates only one connection.

The client-side T Spaces implementation comprises the Tuplespace class, a low-level communication library for sending requests to the T Spaces server, and a ResponseProcessor thread that demultiplexes responses from the server, routing them to the appropriate client thread. Figure 3 depicts a client virtual machine with two application threads. The application threads manipulate instances of the Tuplespace class that use the communication library to send requests to the server.

The communication library assigns each request a unique identifier, which is used to implement a table-based demultiplexing32 of the response stream from the server. The identifier is used as a key to store a request-response synchronization object, the Callback, into the demultiplexing table, the OutstandingRequestTable. The Callback object has two methods: a waitForResponse method that decrements a semaphore, blocking the requesting thread until the response arrives; and a call method that increments the semaphore, thus, unblocking the requesting thread. Figure 4 illustrates the use of the Callback object for synchronization between the requesting thread and the ResponseProcessor.

Using the Callback object as a wrapper for the response-handling code allows all server responses to be handled uniformly. Event-handling code (for registered events), exception-handling services, and poll services (e.g., "are you alive") are all implemented through the Callback object and an OutstandingRequestTable entry.

T Spaces server design. We begin the description of the T Spaces server implementation by presenting a high-level overview of the lifetime of a client Tuplespace operation. In the current model, all client operations act on a single Tuplespace. Reference to or creation of a Tuplespace is made by instantiating a Tuplespace object. Once an instance of a Tuplespace object has been created, the client may execute operations on that Tuplespace. In the current design, a T Spaces server is centralized, and as a result, it does not share data and does not implement multiserver transactions.

Figure 5 illustrates the architecture and shows the layers involved in processing an operation. The operation originates as a method invocation on the Tuplespace object in the client's virtual machine. All the information needed to process the operation is bundled, sent to the server, and unbundled on the server by the client and server communication layers. On the server, the Tuplespace that the operation references is found by the Tuplespace lookup machinery. As we show in the next subsection, this machinery is implemented by simply executing the operation ProcessOperation on the Galaxy Tuplespace that contains tuples representing all the existing Tuplespaces. ProcessOperation finds the correct Tuplespace and passes the operation and tuple operand to that Tuplespace to process.

The server representation of a Tuplespace is the TS class. The TS class encapsulates the tuples in a Tuplespace (the database) and the set of operations that act on the tuples (the factories). The separation of data and the operations that act on the data was a central design goal of the T Spaces architecture; operations are added or changed without affecting the database, and the database is changed without affecting the operations. We discuss the benefits of this design decision in a later subsection.

A "stack" of factories manages the set of operations that act on the Tuplespace. Each factory in the stack maintains a set of handlers that implement specific operations. Given an operation name, a tuple, and a client identifier, a factory returns an implementation of the operation.

This provides maximum flexibility since the factory may custom-tailor the operation implementation to the types of operands and the issuer of the operation. If a factory does not implement an operation, it may request the implementation from the factory immediately below it in the factory stack.

The Tuplespace in Figure 5 has two associated factories. The SimpleDBase factory contains the join and project (projection) handlers, and the Basic factory contains the handlers for the T Spaces API. The SimpleDBase factory maps all join operation strings to its join handler and all project operation strings to its project handler. For other operation strings, it requests the implementation from the Basic factory.

As shown in Figure 5, all operations pass through the TS object that locates the appropriate handler for the operation (from the factory structure), verifies that the issuer of the operation has the correct access control privileges, and, if so, executes the operation using the handler.

Handlers act on a single Tuplespace through the database API of the T Spaces, called the TSDB (Tuplespace database) interface. T Spaces provides both a main memory and a DB2 wrapper implementation of the API via JDBC. The main memory database manages the tuples directly, whereas the DB2 wrapper simply calls DB2 to manage the tuples. Because both types of databases implement the TSDB interface, the same handlers may be used for both. More sophisticated handlers may exploit properties of a specific database type using API calls specific to that database. The database backend provides the core functionality of T Spaces; however, the separation of the operations that operate on the data from the database backend and the layered approach to the design of the overall system provides the flexibility and extensibility of the system.

The current T Spaces server is implemented using four types of threads: ConnectionListener, IOHandler, TSDispatch, and CheckpointManager. The ConnectionListener thread listens for new connections on the connection port number of the server. For each new connection, it creates an IOHandler thread. The IOHandler thread manages all I/O for the newly created client connection. It dispatches incoming client requests by creating a TSDispatch thread to execute the request. Responses are sent by a monitor-protected method in the IOHandler object, which the TSDispatch threads invoke once an operation has been processed. Each Tuplespace has an associated CheckpointManager thread that periodically checkpoints a transaction-consistent state of the Tuplespace. In the following subsections, we describe the individual pieces of the server design of T Spaces.

Server system Tuplespaces. The two system Tuplespaces, Galaxy and Admin, form the catalog of the T Spaces server. The Galaxy space contains tuples describing each Tuplespace that exists on a T Spaces server. Each of these tuples contains the name of the Tuplespace, its type (i.e., main memory or DB2), and a pointer to the internal Tuplespace wrapper object, TS.

The Galaxy Tuplespace is where all operations begin. The ProcessOperation handler of the Galaxy Tuplespace executes an operation on another Tuplespace by looking up the associated Tuplespace in the Galaxy and invoking the operation method of the TS wrapper of that Tuplespace. The Galaxy Tuplespace also implements the CreateTuplespace, DestroyTuplespace, and TuplespaceExists operations.

The Admin Tuplespace contains the access control permissions for each Tuplespace, the groups that each client belongs to, and the user name and password of each client. It is primarily used by the Tuplespace wrapper object, TS, to check whether the submitter of each operation has the proper access control privileges.

T Spaces wrapper object TS. A TS object encapsulates the concept of a collection of tuples and the ability to manipulate that collection. A TS has a name and references to both a database instance (TSDB) and the top-level factory (TSFactory) in its factory stack (recall that the factories map operation names to operation implementations, i.e., handlers).

All operations are processed by the operation method of the TS object. It finds the appropriate handler (the implementation of the operation) for the operation from its factory stack, checks that the client has the access control privileges to execute the handler, and, if so, executes the operation method of the handler object.

The T Spaces server data manager. For flexibility (and scalability) reasons, the T Spaces server allows a different database implementation to be used for each Tuplespace. It is accomplished by associating an instance of a server database with a Tuplespace (as mentioned above) and by allowing different implementations of the abstract interface TSDB. Currently, there are three implementations of TSDB: TSSimpleDB, a simple array-based tuple manager (with simple hash indexes), TSMMDB, a memory-resident data manager (with associated linear hash and T Tree indexes,33 and TSLargeDB, a wrapper and interface for IBM's DB2 system (which provides nearly unlimited storage capacity). We expect that most applications will utilize the lightweight, high-speed main-memory unit, TSMMDB, so we describe it here.

The TSMMDB system evolved from the memory-resident storage management component of the Starburst Relational Database Management System project.34 As shown in Figure 6, a T Space is managed as a collection of fixed-size partitions, each containing a set of tuple references. All references to tuples are kept as data structure addresses rather than real Java language references, which reduces confusion during checkpoints and serialization.

The entire T Spaces server is managed as a "space of spaces" since the Galaxy space contains references to all of the other spaces (including itself) in the server.

T Tree indexes currently come in two types: the Modified Linear Hash index (for exact match only) and the T Tree index (for range and ordered matches). The Modified Linear Hash index35 is a simple main-memory variation upon Withold Litwin's Linear Hash algorithm36 and is not described here. Similarly, the T Tree index structure is a fairly simple data structure, which we briefly describe here.

The T Tree data structure combines the efficient lookup characteristics of an AVL tree (named from the initials of its three developers) with the space efficiency of a B tree.33,35 It is relatively easy to implement, and because it is a memory-resident data structure, it allows a wide variety of items to be indexed in a variety of ways. Since the items in the T Tree are references, and not actual data, the T Tree does not need to manage variable-size data. Also, since the references can point to virtually anything, the T Tree function is really determined by the compare routine, which can perform any type of compare that an implementer creates. These include multiattribute indexes, multidimensional indexes, inverted text indexes, and, of course, "regular" single-attribute indexes.

Figure 7 shows an instance of a T Tree node (a close-up) and a T Tree. A T

Tree is a multi-item node AVL tree, with some special tree-balancing operations that are needed to deal with the multi-item nodes. A T Tree node contains an array of data references, a parent pointer, two child pointers, and some control information.

Factories and handlers: Customizing operations. Using the object-oriented design pattern of factories and handlers,37 T Spaces employs factories to track and manipulate handlers, the implementations of all T Spaces operations. The factory-handler approach allows the databases of T Spaces to be dynamically customized. New operations may be added to increase functionality, existing operation implementations may be changed without halting the system, and operation implementations may be customized for their operands or for the client issuing the operation, or both. Given an operation string, a tuple operand, and the client identifier for the issuer of the operation, a factory selects an implementation (a handler) to execute the operation.

A T Spaces operator is associated with a particular operator family. For example, in the current implementation of T Spaces there are three families: Tuplespace operations, Administrative operations, and Galaxy operations. For each operator family, there is a stack of one or more factories that map the operators to handlers. If a factory does not map a given operation string, operand, and client identifier to a handler, it may query the factory below it in the stack for a handler for the operation. A factory may block an implementation of an operation by simply throwing a HandlerNotFound exception when it is queried for the operation. Only the top-level factory on the stack is directly accessible; it uses the lower-level factories when it does not implement or block an operation string, operand, and client identifier triple. A factory must implement the TSFactory interface, which has methods for allowing or disallowing the stacking of other factories on top of it, and for returning a handler given an operation string, tuple operand, and client identifier triple.

Although it is not a trivial exercise, educated users can write sophisticated handlers and download them into the T Spaces server for execution. All handlers must implement the TSHandler interface, which has a method for executing an operation, given the tuple operand of the operation. Handler implementers have available to them the Database API of T Spaces, which allows them to move tuples into and out of storage and to perform data-related operations on them. New factories and handlers may be downloaded to the server using the addFactory and addHandler methods, respectively. Of course, extreme care must be taken when writing new factories and handlers, and only trusted users should be allowed to download them to the server. In the current T Spaces prototype, only designated system administrators have the access control privileges required to add new factories and handlers to a Tuplespace.

Durability. The T Spaces server must ensure that the effects of committed transactions are durable. All of the operations of a transaction are recorded in a redo log38 on stable storage before the transaction commits. In the event of a server failure, the log is used to replay the operations of committed transactions. Periodically, each Tuplespace database is written to stable storage, and its log is truncated. In addition to making tuple values durable, the T Spaces server must also make uploaded types durable.

New types arrive at the server in the form of tuple fields or as subclasses of SubclassableTuple. In both cases, the type needs to be saved on stable storage before the transaction that uploaded it commits. Java remote method invocation (RMI), used by JavaSpaces, provides a transparent mechanism for loading types from client to server and server to client; however, it does not make those types persistent. Because we needed to create our own mechanisms to make uploaded types persistent and because transmitting data with Java RMI is slower than with Java sockets,39 we chose to build our own uploading mechanism using Java sockets and ClassLoaders.40

For each new client connection, a ClassLoader is created (the ClientClassLoader) and is used to create the IOHandler for the connection. The Java run-time system then uses the ClientClassLoader to load any classes needed during the execution of that instance of the IOHandler class. Specifically, when a tuple that contains new types is uploaded from a client, the ClientClassLoader for that client connection is used to upload the bytecode for the new type. The server tags these types with the URL (uniform resource locator) of the client that uploaded them so that if they reference other types, the server can upload those types, provided the client is still connected. On the client side, a ClassLoader for each server connection is used to download types from the servers.

Access control. In the spirit of the Andrew File System (AFS),41 the T Spaces access control architecture is hierarchical. Each Tuplespace in the hierarchy defines a group-based set of access control permissions that govern the creation and deletion of "children" Tuplespaces, reading and writing of tuples, and event registration. In this way access control can be enforced on the Tuplespace level; finer granularity is not supported. This architecture allows T Spaces users to create their own "domains" that they can administer themselves. Figure 8 depicts an access control hierarchy (note that this logical hierarchy applies to access control only; in particular, it does not apply to the storage and retrieval of tuples in Tuplespaces themselves). In the figure, the Tuplespace "grand central" is a child of the "IBM" Tuplespace. Its administrator has created a local user "John" and a local group "GCS" to which "John" has been added. The user "John" and group "GCS" are visible to the "grand central" and "cyber-soda" Tuplespaces but not to the "IBM" or "garlic" Tuplespaces. As demonstrated by AFS, providing a hierarchy of access control administration allows a system to be scalable with respect to both administration and implementation. Even for our current single-server architecture, allowing users to create their own "domains" of access control is useful, and for larger-scale deployment, it is a necessity.

The access control architecture is composed of the user and group administration, handler permissions, and Tuplespace access control administration components. Figure 9 shows how these three components determine the access permissions for each operation defined on a Tuplespace. As the figure shows, the handlers publish a set of "AccessAttributes" such as "READ" that define their access control requirements. The Tuplespace access control component defines on a per-Tuplespace basis which groups have permission for which "AccessAttribute," and the user and group component manages user names, passwords, and group information. Caching of access permissions is stored in the IOHandler object on a per-connection basis.

The access control mechanism may also be attached to any existing system so that users can merge the T Spaces access control mechanism with that of a file system or database system. The T Spaces server allows system administrators to overload the user and group lookup routines so that they call the existing system rather than the internal T Spaces routines.

T Spaces events. In T Spaces, an event is defined as the execution of an operation on a T Space (exceptional conditions are not events). Any event may be registered for using the eventRegister method. Besides the T Spaces, and the operation and tuple template that specify the event of interest, the client also specifies an object that implements the Callback interface. When the event occurs, the system automatically calls the call method of that object. In the logical implementation of events, when an event for which there are clients registered occurs, "event notification tuples" are created for each registration. These tuples are created in the transaction that created the event of interest, and thus, their creation will be atomic with the commit of the operation that is the event.

Application examples

Potential applications for T Spaces come in all shapes and sizes, partly because T Spaces serve a very general service and partly because they can be used in several different ways. We have already mentioned the two main uses, namely, as a communication buffer or as a database manager. However, it is also useful as a synchronization mechanism (either by clients registering for events or just waiting for signal tuples to arrive) and for central control of real-time events, such as communication (chat rooms) or multiplayer game controllers.

In this section, we present three different sample applications. The first two, the distributed clipboard and the shared whiteboard, we implemented easily. The third application presented here is an exercise in how we might apply T Spaces to a real-world problem.

The Global Cut and Paste Buffer. In its simplest form, a T Space acts as a "global whiteboard" for communication among a group of distributed processes. A process with data to share with the group simply writes a tuple into a Tuplespace. A process looking for data reads a tuple by providing a matching template. In this section, we demonstrate this basic use with the Global Cut and Paste Buffer. This application allows a user to "publish" the contents of his or her local clipboard for viewing by any other user on the network. Publishing enables easy and flexible sharing of any data recognized by Java.

This application demonstrates the ease with which distributed applications can be written using T Spaces. Once a Local Cut and Paste Buffer has been written, changing it into a Global Cut and Paste Buffer is trivial. We implemented our own Global Cut and Paste Buffer, the GCS Clipboard Viewer, in which each user's "buffer space" contains four text buffers and is stored in a Tuplespace corresponding to that user's name. Buffer i is represented as the tuple (i , buffer value). To view a buffer space, the user enters the name of the space, and the application simply creates an instance of that user's Tuplespace (i.e., Tuplespace ts = new Tuplespace (user name)), and reads each buffer tuple (e.g., Tuple buffer = ts.read(1 , field.makeFormal("String"))). To publish a buffer space, the application performs takes on the old buffer tuples and replaces them with new buffer tuples using the write operation. Only users with the proper access control permissions may read and write these Tuplespaces.

Figure 10 shows a snapshot of our clipboard viewer. In this figure, the content of buffer one indicates that one of the authors, Toby Lehman, is publishing the URL for a San Jose Mercury News story; in buffer two he is publishing an e-mail snippet that he wants to copy onto another machine in his office; and in buffer three he is exporting a letter written in support of a Harvey Mudd College student. Once Toby's buffer space has been published on the T Spaces server, other users can use their T Spaces clipboard viewers (assuming the proper security privileges) to view the contents. This example shows that T Spaces provides a simple, easy-to-use mechanism for building simple distributed applications.

The distributed whiteboard. T Spaces may also provide a distributed whiteboard for communication among a group of distributed processes. A process that wants to modify part of the whiteboard simply updates tuples in the whiteboard space. A process wishing to see the whiteboard may read the tuples in the whiteboard space. We implemented a simple line drawing whiteboard using T Spaces. Each client runs a Java applet that continuously refreshes its view of the whiteboard either by reading the tuples in the bulletin board space or registering for update events on the whiteboard space. Each line in the whiteboard is represented as a tuple. The user may delete existing lines or add new lines. When the user adds lines, the application simply creates a tuple for each new line and sends it to the T Spaces server. To delete lines, the application creates templates for the lines to be deleted and simply performs a take with those templates. The changes are immediately available to other users. Figure 11 shows what a user sees while editing a figure using the GCS whiteboard. Once the user is finished editing, the application will create tuples for newly created lines, send those to the T Spaces server, create template tuples for each destroyed line, and send take requests for the destroyed line tuples to the T Spaces server. These small example applications show that T Spaces is simple and easy to use and thus well-suited for simple applications. In the next subsection, we describe how T Spaces could be used for a more complex application.

The ubiquitous computing environment. Since T Spaces connects all programs to all programs, it also connects all programs to all services that are supported by programs. For example, a collection of printers attached to a UNIX** LAN (local area network) would be visible to any program running on any client, without any prior printer definition. Similarly, scan, fax, e-mail and pager services would be available to arbitrary clients via T Spaces, as well as any general program service, such as a search engine or image translation program.

In the T Spaces world, a common computing environment with access to all possible network services is surprisingly easy to build. A set of applications, written in Java, map the system-specific service (e.g., printing service) or application (e.g., Web search) to a standard tuple representation. Any client from any platform can generate a tuple and send it to a T Spaces server. Applications, listening for specific tuples, pick up jobs when they see them and execute them. In building this environment, one would write the service application to run on the platforms where they could do the most good. At the IBM Almaden Research Center, most network printers are visible to AIX systems. Therefore, one AIX printer service application would listen for tuples posted to the "printing space," pick them up, and send them to an AIX queue. Similarly, queries about queues and print status could also be done. On every other local printer that should be attached to the common platform, a user simply runs the local print application (although, in practice, the local application can probably handle a number of services). We anticipate little system interruption when deploying this scheme, since the T Spaces global environment is only additive, not disruptive. Thus, to an existing computing environment, T Spaces simply adds connectivity.

One of the more advanced aspects of the T Spaces global environment is that it allows the introduction of a smart middleman into the equation. Therefore, as mentioned above, one can build a useful independent print service that provides access to any printer in the building, which is beneficial, as long as the whereabouts of all the printers is known. However, a smart middleman can add intelligence. A PDA with a location sensor gives the current location to the smart middleman, which then in turn selects the most appropriate printer, based on location, availability, speed, resolution, etc. In addition, after having chosen a printer, the middleman can also display a map to the PDA user showing the path from the user's current location to the printer.

Just as a printing service can be made generally available through T Spaces, so can any other service. But the real power of the environment is shown by how new computers, such as the Tier-0 devices, are enabled with any network service and are connected to any other network computer through the simple T Spaces interface. Data from a smart cell phone can easily be made available to a PDA, an automobile computer, a home security computer, a personal computer, or even a corporate mainframe computer.

Conclusion, project status, and future work

It is interesting that people often accept limitations without question when no alternatives exist but then will not tolerate anything but the best solution when better alternatives are known. Currently, the model for distributed programming is synchronous, directed point-to-point communication, and the programming community appears to accept this without question. Although the Tuplespace model of communication has numerous benefits for the distributed application programmer, most Tuplespace efforts of the past remain unknown to the general population, having been known only to the relatively obscure area of parallel programming. However, with T Spaces and Sun Microsystems' JavaSpaces pushing into the mainstream of computing, this is about to change. The usefulness of the Tuplespace model is simply too promising to ignore, and, with the added diversity of the emerging Tier-0 devices breaking the existing point-to-point communication model, the timing could not be more perfect.

T Spaces embodies the convenience of the Tuplespace communication model, the robust reliability and trustworthiness of a database system, and the platform independence of the Java programming language. It contains most of the features that today's Tier-0 devices need in a network middleware package. Furthermore, it not only serves as the interface to other programs and distributed components, it is a fully functional data repository that can act as the primary data store or a caching front end to a corporate data warehouse.

The T Spaces project released version 1.02 to the public via the IBM AlphaWorks* channel (the URL for AlphaWorks is http://www.alphaworks.ibm.com) on March 16, 1998. The release contained most of the features described in this paper, except for the recovery features, which, at that time, were still in development. As part of the next release, version 2, we will be adding function and support for additional interfaces to the T Spaces server. The T Spaces engine itself will be improved with features like XML (Extensible Markup Language) support so that XML data can be indexed and searched, a distributed beans interface so that JavaBeans** can listen for, and receive, events on distributed machines, a CORBA interface so that T Spaces can communicate directly with non-Java platforms, and a mobile interface so that T Spaces can function as a replication server to the mobile clients.

We have not yet run any formal performance tests on T Spaces, only small-scale informal ones. We are optimistic with results so far in terms of search, insert, and delete performance. However, a true multiuser experiment that tests T Spaces performance and scalability under significant user load will provide truly useful results.

Acknowledgments

We thank the IBM Grand Central Station team for their extensive testing of our system, the Harvey Mudd College team for their work on the Palm Pilot** Client, and Norm Pass, for his continued support of our project. Our special thanks to John Thomas for getting our system ready for public release and thanks also to Kevin Eustice, programmer extraordinaire, for his fine proofreading and coding capabilities.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Apple Computer Inc., Object Management Group, Oracle Corp., Sybase, Inc., Informix Software, Inc., Sun Microsystems, Inc., The Open Group, or 3Com Corp.

Cited references and notes

1. Popular Electronics (January 1975).

2. http://www.omg.org/about/wicorba.htm, Object Management Group.

3. J. M. Cheng, C. R. Loosley, A. Shibamiya, and P. S. Worthington, "IBM Database 2 Performance: Design, Implementation, and Tuning," IBM Systems Journal 23, No. 2, 189-210 (1984).

4. http://www.software.ibm.com/data/db2/udb, IBM Corporation.

5. R. Bamford, "Oracle: A High-Performance Production DBMS," presentation at 19th ACM SIGMOD Conference on the Management of Data (May 1987).

6. http://www.oracle.com, Oracle Corporation.

7. P. Melmon, "The Sybase Open Server," Proceedings of the SIGMOD nternational Conference on Management of Data, SIGMOD Record 21, No. 2, 413-413 (June 1992).

8. http://www.sybase.com, Sybase, Inc.

9. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," Communications of the ACM 34, No. 10, 50-63 (October 1991).

10. http://www.objectdesign.com, Object Design Co.

11. Y.-M. Shyy, "VERSANT Replication: Supporting Fault-Tolerant Object Databases," Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (May 1995), pp. 441-442.

12. http://www.versant.com, Versant Co.

13. Building an Object-Oriented Database System--The Story of O2, F. Bancilhon, C. Delobel, and P. Kanellakis, Editors, Morgan Kaufmann Publishers, San Francisco, CA (1992).

14. http://www.o2tech.fr, O2 Technology, Ardent Software, Inc.

15. R. Ananthanarayanan, V. Gottemukkala, W. Kafer, T. J. Lehman, and H. Pirahesh, "Using the Co-Existence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems," Proceedings of

the ACM SIGMOD Conference (1993), pp. 109-118.

16. B. Reinwald, S. Dessloch, M. J. Carey, T. J. Lehman, H. Pirahesh, and V. Srinivasan, "Making Real Data Persistent: Initial Experiences with SMRC," POS (1994), pp. 202-216.

17. S. J. White and D. J. DeWitt, "QuickStore: A High Performance Mapped Object Store," SIGMOD Record 23, No. 2, 395-406 (June 1994).

18. M. Stonebraker, "Postgres DBMS," SIGMOD Record 19, No. 2, 394 (June 1990).

19. R. Tolksdorf, "Laura: A Coordination Language for Open Distributed Systems," 13th IEEE International Conference on Distributed Computing Systems (1993), pp. 39-46.

20. A. Polze, "Using the Object Space: A Distributed Parallel Make," 4th IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon (September 1993), pp. 234-239.

21. P. Ciancarini, A. Knoche, D. Rossi, R. Tolksdorf, and F. Vitali, "Coordinating Java Agents for Financial Applications on the WWW," Proceedings of the 2nd Conference on Practical Applications of Intelligent Agents and MultiAgent Technology (PAAM) (April 1997), pp. 179-193.

22. JavaSpace Specification, Revision 0.4, http://java.sun.com:80/products/javaspaces/specs/js.ps, Sun Microsystems, Inc.

23. P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali, "PageSpace: an Architecture to Coordinate Distributed Applications on the Web," Computer Networks and ISDN Systems 28, No. 7-11 (May 1996).

24. L. Stesin, personal communication (July 1997).

25. B. Anderson and D. Shasha, "Persistent Linda: Linda + Transactions + Query Processing," Workshop on Research Directions in High-Level Parallel Programming Languages, Mont-Saint-Michel, France (June 1991). Published as Springer-Verlag Lecture Notes in Computer Science 574.

26. N. Afshartous and M. C. Harrison, "Expressing Concurrency in Griffin," IEEE International Conference on Parallel and Distributed Systems (1996), pp. 292-301.

27. D. Gelernter, "Generative Communication in Linda," TOPLAS 7, No. 1, 80-112 (1985).

28. N. Carriero and D. Gelernter, "Linda in Context," Communications of the ACM 32, No. 4, 444-458 (April 1989).

29. D. Gelernter and A. J. Bernstein, "Distributed Communication via Global Buffer," Proceedings of the ACM Principles of Distributed Computing Conference (1982), pp. 10-18.

30. In the original Linda systems, the fields were restricted to primitive types such as integer and string, and aggregates such as structures and arrays. Several modern systems, including ours, eliminate this restriction.

31. In fact, by attaching a Structured Query Language (SQL) front end to T Spaces, we could create a Java-based SQL database engine.

32. D. Lea, Concurrent Programming in Java Design Principles and Patterns, Second Edition, Addison-Wesley Publishing Co., Reading, MA (1997).

33. T. J. Lehman and M. J. Carey, "Query Processing in Main Memory Database Management Systems," Proceedings of the ACM SIGMOD Conference (1986), pp. 239-250.

34. T. J. Lehman, E. J. Shekita, and L. F. Cabrera, "An Evaluation of Starburst's Memory Resident Storage Component," Transactions on Knowledge and Data Engineering 4, No. 6, 555-566 (1992).

35. T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," Proceedings of the IEEE International Conference on Very Large Data Bases (1986), pp. 294-303.

36. W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," Proceedings of the IEEE International Conference on Very Large Data Bases (1980), pp. 212-223.

37. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Co., Reading, MA (1994).

38. B. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley Publishing Co., Reading, MA (1987).

39. S. Hirano, Y. Yasu, and H. Igarashi, "Performance Evaluation of Popular Distributed Object Technologies for Java," ACM Workshop on Java for High-Performance Network Computing (February 1998), pp. 81-90.

40. A ClassLoader is the piece of the Java run-time system that is responsible for finding a class given its name. The ClassLoader that loads a class is used to find any classes that the loaded class needs during its execution.

41. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," ACM Transactions on Computer Systems 6, No. 1, 51-81 (February 1988).

Accepted for publication April 6, 1998.

Peter Wyckoff

Courant Institute of Mathematical Sciences, New York University, Computer Science Department, 251 Mercer Street, New York, New York 10012 (electronic mail: wyckoff@cs.nyu.edu). Mr. Wyckoff is pursuing a Ph.D. in computer science at New York University. He holds a bachelor's degree from the State University of New York at Stony Brook and a master's degree from New York University. He participated in the T Spaces project while he was an intern at the IBM Almaden Research Center.

Stephen W. McLaughry

University of Oregon, Computer and Information Science Department, Eugene, Oregon 97403 (electronic mail: stephen@cs.uoregon.edu). Mr. McLaughry is pursuing a Ph.D. in computer science at the University of Oregon. His research interests include type theory and distributed systems. He holds a bachelor's degree from Williams College and a master's degree from the University of Oregon. He worked on the T Spaces project while he was an intern at the IBM Almaden Research Center.

Tobin J. Lehman

IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (electronic mail: toby@almaden.ibm.com). Dr. Lehman joined the Almaden Research Center in 1986, shortly after receiving his Ph.D. from the University of Wisconsin-Madison. His thesis on the area of memory-resident database systems introduced some novel concepts in index structures, query processing, logging, recovery, and concurrency control. At IBM he participated in a number of projects, including distributed database systems (R*), a hierarchical database machine, a new age computing environment, an extensible database system project (Starburst), an object-relational database system (SMRC), and a revolutionary Web search/publish system (Grand Central Station). Dr. Lehman was one of the original architects of the IBM ADSM backup product, and he was the creator and chief implementer of both the text search functions and the Large Object (LOB) support in the DB2 Common Server Version 2. He is currently the leader of the T Spaces project in the computer sciences department. His research interests include object-relation database systems, large object management, memory-resident database systems, making the world smaller through indexing the World Wide Web, and using T Spaces to increase productivity.

Daniel A. Ford

IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (electronic mail: daford@almaden.ibm.com). Dr. Ford manages the Grand Central Station research project in the computer sciences department. His research interests include Web crawling, Web channel publication, XML standards, RAID tertiary storage, intelligent home networks, and knowledge management.

Reprint Order No. G321-5686.

Table 1 Simple tuple examples

Sample Tuple	Description	Does the sample match the template (float, "hello world", int)?	D sam the (flo
<2.24, "hello world", 345>	A tuple with three fields: (1) a float with the value 2.24, (2) a string with the value "hello world", and (3) an integer with the value 345.	Yes	
<2.24, "hello world", 345.0>	A tuple with three fields: (1) a float with the value 2.24, (2) a string with the value "hello world", and (3) a float with the value 345.0.	No	

< >	A tuple with ( ) fields.	No

Table 2 Tuple object matching examples; Person is a subclass of SubclassableTuple and Employee is a subclass of Person.

Sample Tuple	Description	Does the sample match th template Employee(1089921 "John", "Doe")?
Tuple ("hello")	A structural equivalence only tuple	No
Person (108992145, "John", "Doe")	A Person tuple with three fields: "SS#", "Fname", and "Lname"	No (an object of type Person cannot implement Employee object)
Employee (108992145, "John", "Doe")	An Employee (a subclass of person) with three fields: "SS#", "Fname", and "Lname"	Yes

Table 3 Query examples

Query #	Query Type	Query Example
1	Regular query	`resultTuple = ts.read("Superman", 75, new Rock("Kryp`

2	Match queries	`queryTuple = new Tuple("Superman", 75, Rock );`
		`resultSetTuple = ts.scan(new MatchQuery( queryTuple`

3	Index queries	`queryTuple = new Tuple(new IndexQuery("Superheros",`
		`resultSetTuple = ts.scan( queryTuple );`

4	Range queries	`queryTuple = new Tuple(new (IndexQuery("Superheros",`
		`resultSetTuple = ts.scan( queryTuple );`

5	And queries	`queryTuple = new Tuple( new AndQuery(`
		`new IndexQuery("Superheros", new Range(`
		`new IndexQuery("Age", new Range(new Int`
		`new Integer(30)));`
		`resultSetTuple = ts.scan(queryTuple);`

6	Or queries	`queryTuple = new Tuple( new OrQuery(`
		`new IndexQuery("Age", new Range(new Int`
		`new IndexQuery("Age", new Range(new Int`
		`resultSetTuple = ts.scan(queryTuple);`